Practical – 1

Aim: Implement and analyse algorithms given below:

1.1) Factorial (Iterative and Recursive)

1.2) Fibonacci Series(Iterative and Recursive)

1.3) GCD (Iterative and Recursive)

Program Code:

```
import java.util.*;
public class master
{
   public static int factorial(int n)//recursive factorial
   {
      if(n==1)
      {
         return 1;
      }
      return (n*factorial(n-1));
   }
   public static int fact(int n)//iterative factorial
   {
      int fact=1;
      for(int i=1 ; i<=n ; i++)
      {
         fact=fact*i;
      }
      return fact;
   }
   public static int fib(int n)//recursive fibonacci
```

```
{
   if(n==0 || n==1)
   {
      return n;
   }
   return (fib(n-1)+fib(n-2));
}
public static int ffib(int n)//iterative fibonacci
{
   int pprev, prev = 0, curr = 1;
   for (int i = 1; i < n ; i++)
   {
      pprev = prev;
      prev = curr;
      curr = pprev + prev;
   }
   return curr;
}
public static void main(String args[])
{
   long begin_time=System.currentTimeMillis();
   System.out.println("factorial of 5= "+factorial(5));
   long end_time=System.currentTimeMillis();
   System.out.println("time takenby recursive approach= "+(end_time-begin_time));


   long ibegin_time=System.currentTimeMillis();
   System.out.println("factorial of 5= "+fact(5));
```

```
        long iend_time=System.currentTimeMillis();
        System.out.println("time takenby iterative approach= "+(iend_time-ibegin_time));

        long fbegin_time=System.currentTimeMillis();
        System.out.println("fibonacci series "+fib(7));
        long fend_time=System.currentTimeMillis();
        System.out.println("time taken by recursive approach= "+(fend_time-fbegin_time));

        long tbegin_time=System.currentTimeMillis();
        System.out.println("fibonacci series upto 5= "+ffib(6));
        long tend_time=System.currentTimeMillis();
        System.out.println("time taken by iterative approach= "+(tend_time-tbegin_time));
    }
}
import java.util.*;
public class k
{
    //Iterative GCD
    public static int gcd(int a, int b)
    {
     int result = Math.min(a, b);
     while (result > 0) {
         if (a % result == 0 && b % result == 0) {
             break;
         }
         result--;
    }
        return result;
```

```
    }
    //Recursive GCD
    public static int gcd2(int a, int b)
    {
        if (a == 0)
            return b;
        if (b == 0)
            return a;
        if (a == b)
            return a;
        if (a > b)
            return gcd(a - b, b);
        return gcd(a, b - a);
    }
    public static void main(String args[])
    {
        long begin_time=System.currentTimeMillis();
        System.out.println("GCD Answer= "+gcd(5,15));
        long end_time=System.currentTimeMillis();
        System.out.println("time taken by GCD= "+(end_time-begin_time));

        long ibegin_time=System.currentTimeMillis();
        System.out.println("GCD Answer= "+gcd2(5,15));
        long iend_time=System.currentTimeMillis();
        System.out.println("time taken by GCD= "+(iend_time-ibegin_time));
    }
}
```

OUTPUT:

	Sr.No.	Iterative(in seconds)	Recursive(in seconds)
Factorial	1	0.014	0.014
	2	0.017	0.021
	3	0.014	0.013
	4	0.015	0.015
	5	0.019	0.019

	Sr.No.	Iterative(in seconds)	Recursive(in seconds)
Fibonacci	1	0.013	0.012
	2	0.017	0.014
	3	0.0112	0.028
	4	0.215	0.567
	5	0.013	0.105

	Sr.No.	Iterative(in seconds)	Recursive(in seconds)
GCD	1	0.014	0.024
	2	0.015	0.021
	3	0.014	0.015
	4	0.014	0.029
	5	0.019	0.026

CONCLUSION :For factorial function both iterative and recursive approach behaves same. For Fibonacci series iterative approach is better because as n increases recursive algorithm has exponential increase in time complexity.For

GCD iterative approach is better and preferrable because as values increases recursive approach is inefficient and takes more elapsed time for computation.

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 2

Aim: Implement and analyse algorithms given below:

2.1) Binary Search

2.2) Insertion Sort

Program Code:

```
import java.util.*;
import java.time.*;
public class trial
{
   public static int BinarySearch(int a[],int key)
   {
      int start=0;
      int end=a.length-1;
      while(start<=end)
      {
         int mid=start+(end-start)/2;
         if(a[mid]==key)
         {
            return mid;
         }
         else if(key>a[mid])
         {
            start=mid+1;
         }
         else
```

```
        {
            end=mid-1;
        }
    }
    return -1;
}
public static void InsertionSort(int a[])
{
    for(int i=1 ; i<a.length ; i++)
    {
        int temp=a[i];
        int j=i-1;
        while(j>=0 && temp<a[j])
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=temp;
    }
}
public static void main(String[] args)
{
    int key=30;
    int a[]={10,20,30,40,50,60,70,80,90,100};
    System.out.println();
```

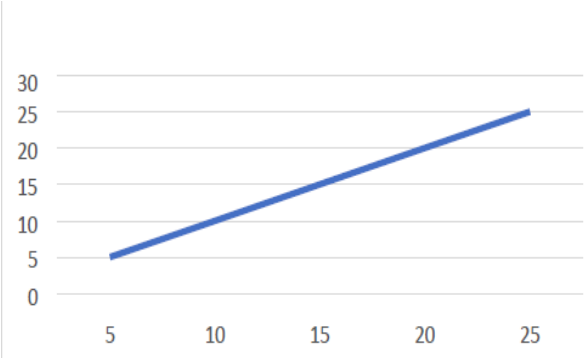
```
        long begin_time=System.currentTimeMillis();
        System.out.println(key+" is found at "+BinarySearch(a,key)+" using Binary Search\n");
        long end_time=System.currentTimeMillis();
        System.out.println("time taken by Binary Search= "+(end_time-begin_time));


        System.out.println("\nQuick Sort");
        int b[]={5,2,6,7,1,3,4};
        Instant inst11 = Instant.now();
        InsertionSort(b);
        Instant inst22 = Instant.now();
        System.out.println("Elapsed Time: "+ Duration.between(inst11, inst22).toString());
        display(b);
}
```

OUTPUT:

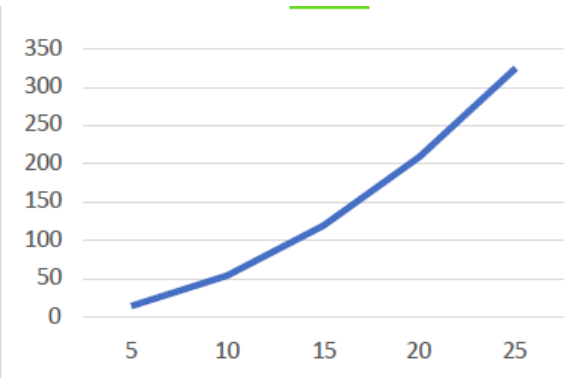
```
PROBLEMS 92  OUTPUT  DEBUG CONSOLE  PORTS  TERMINAL  COMMENTS

PS D:\Probin's Work\DSA> javac mytree.java
PS D:\Probin's Work\DSA> java mytree

30 is found at 2 using Binary Search

time taken by Binary Search= 6 ms
```

```
Insertion Sort
Elapsed Time: PT0.000004795S
Array:
 1 2 3 4 5 6 7
```

Insertion Best Case



Insertion Average Case



CONCLUSION: From this practical I learned about binary searching which is faster than linear search which is suitable for large datasets. Time Complexity is O(log$n$). and insertion sort technique which is adaptive in nature which is suitable for datasets that is already partially sorted. Time Complexity is O(n^2).

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 3

Aim: Implement and analyse algorithms given below:

3.1) Implement and analyze Quick Sort Algorithm

Program Code:

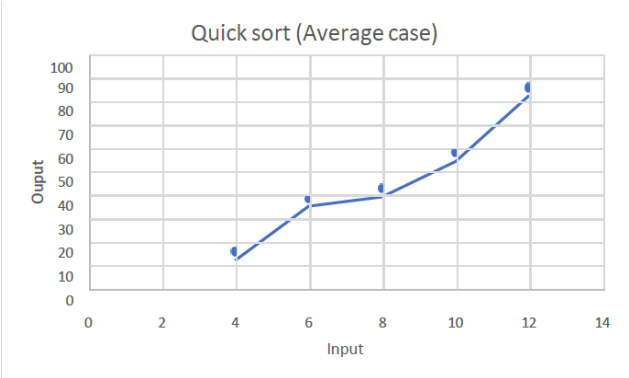
```
import java.util.*;
import java.time.*;
public class trial
{
   public static void QuickSort(int a[],int starting,int ending)
   {
      if(starting>=ending)
      {
         return;
      }
      int pivot_index=partition(a,starting,ending);
      QuickSort(a, starting, pivot_index-1);
      QuickSort(a, pivot_index+1, ending);
   }
   public static int partition(int a[],int starting,int ending)
   {
      int pivot=a[ending];
      int i=starting-1;
      for(int j=starting ; j<=ending ; j++)
      {
         if(a[j]<pivot)
```

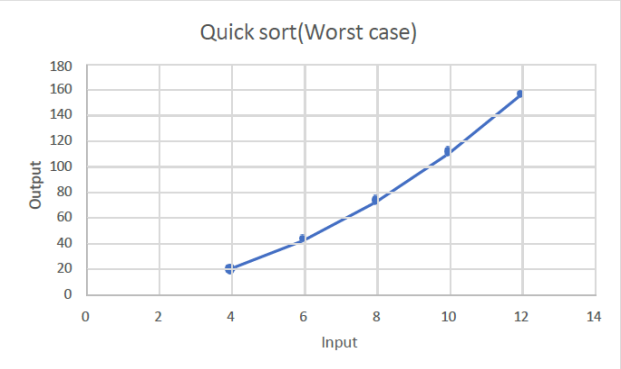
```
            {
                i++;
                int temp=a[j];
                a[j]=a[i];
                a[i]=temp;
            }
        }
        i++;
        int temp=pivot;
        a[ending]=a[i];
        a[i]=temp;
        return i;
    }
    public static void display(int a[])
    {
        System.out.println("Array: ");
        for(int i=0 ; i<a.length ; i++)
        {
            System.out.print(" "+a[i]);
        }
        System.out.println();
    }
public static void main(String[] args)
    {
        System.out.println("\nQuick Sort");
```

```
        int c[]={5,2,6,7,1,3,4};
        Instant inst1 = Instant.now();
        QuickSort(c,0,c.length-1);
        Instant inst2 = Instant.now();
        System.out.println("Elapsed Time: "+ Duration.between(inst1, inst2).toString());
        display(c);}
}
```

Output:

```
Quick Sort
Elapsed Time: PT0.000009024S
Array:
 1 2 3 4 5 6 7
```





Conclusion: From this practical I learned about quick sort technique and how it used divide and conquer strategy for sorting the data. Time complexity is O(nlogn).Quick Sort technique is efficient on large datasets.

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 4

Aim: Greedy Approach

4.1) A Burglar has just broken into the Fort! He sees himself in a room with n piles of gold dust. Because each pile has a different purity, each pile also has a different value (v[i]) and a different weight (w[i]). A Burglar has a bag that can only hold W kilograms. Calculate which piles Burglar should completely put into his bag and which he should put only fraction into his bag. Design and implement an algorithm to get maximum piles of gold using given bag with W capacity, Burglar is also allowed to take fractional of pile.

Program Code:

```
import java.util.*;
public class prac {


    private static class Item {
        int value, weight;
        double ratio;


        public Item(int value, int weight) {
            this.value = value;
            this.weight = weight;
            this.ratio = (double) value / weight;
        }
    }
    private static void sortItems(Item[] items) {
        for (int i = 0; i < items.length - 1; i++) {
            for (int j = i + 1; j < items.length; j++) {
                if (items[i].ratio < items[j].ratio) {
```

```
                Item temp = items[i];
                items[i] = items[j];
                items[j] = temp;
            }
        }
    }
}
private static double getMaxValue(int[] values, int[] weights, int capacity) {
    int n = values.length;
    Item[] items = new Item[n];
    for (int i = 0; i < n; i++) {
        items[i] = new Item(values[i], weights[i]);
    }
    sortItems(items);
    double totalValue = 0;
    int currentWeight = 0;
    for (Item item : items) {
        if (currentWeight + item.weight <= capacity) {
            totalValue += item.value;
            currentWeight += item.weight;
        } else {
            int remainingCapacity = capacity - currentWeight;
            totalValue += (double) item.value * remainingCapacity / item.weight;
            break;
        }
    }
    return totalValue;
```

```
    }
    public static void main(String[] args) {
        int[] values = {60, 100, 120};
        int[] weights = {10, 20, 30};
        int capacity = 50;
        double maxValue = getMaxValue(values, weights, capacity);
        System.out.println("Maximum value of gold dust that the burglar can steal: " +
maxValue);
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Maximum value of gold dust that the burglar can steal: 240.0
PS D:\Probin's Work\Extra>
```

4.2) Implement the program to find the shortest path from one source to all other destinations in any city graph.

Program Code:

```
import java.util.*;
public class prac {
   private int V;
   private List<List<Node>> adj;
   public prac(int V) {
      this.V = V;
      adj = new ArrayList<>(V);
      for (int i = 0; i < V; i++) {
         adj.add(new ArrayList<>());
```

```
    }
}
class Node implements Comparable<Node> {
    int dest;
    int weight;

    Node(int dest, int weight) {
        this.dest = dest;
        this.weight = weight;
    }

    public int compareTo(Node other) {
        return Integer.compare(this.weight, other.weight);
    }
}
public void addEdge(int source, int dest, int weight) {
    adj.get(source).add(new Node(dest, weight));
    adj.get(dest).add(new Node(source, weight));
}

public void shortestPath(int source) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    int[] dist = new int[V];
    Arrays.fill(dist, Integer.MAX_VALUE);
    pq.add(new Node(source, 0));
    dist[source] = 0;
```

```
    while (!pq.isEmpty()) {
        int u = pq.poll().dest;
        for (Node neighbor : adj.get(u)) {
            int v = neighbor.dest;
            int weight = neighbor.weight;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.add(new Node(v, dist[v]));
            }
        }
    }
    System.out.println("Shortest paths from source " + source + " to all destinations:");
    for (int i = 0; i < V; i++) {
        System.out.println("Destination " + i + ": " + dist[i]);
    }
}
public static void main(String[] args) {
    int V = 5;
    prac graph = new prac(V);
    graph.addEdge(0, 1, 2);
    graph.addEdge(0, 3, 6);
    graph.addEdge(1, 2, 3);
    graph.addEdge(1, 3, 8);
    graph.addEdge(1, 4, 5);
    graph.addEdge(2, 4, 7);
    graph.addEdge(3, 4, 9);
```

```
    graph.shortestPath(0);
  }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Shortest paths from source 0 to all destinations:
Destination 0: 0
Destination 1: 2
Destination 2: 5
Destination 3: 6
Destination 4: 7
PS D:\Probin's Work\Extra>
```

4.3) Find Minimum Cost spanning tree of an undirected graph using Kruskal's algorithm.

Program Code:

```
import java.util.*;
class Edge {
  int source, dest, weight;

  Edge(int s, int d, int w) {
    source = s;
    dest = d;
    weight = w;
  }
}
class Graph {
```

```
int V, E;
Edge[] edges;

Graph(int v, int e) {
    V = v;
    E = e;
    edges = new Edge[E];
    for (int i = 0; i < e; ++i) {
        edges[i] = new Edge(0, 0, 0);
    }
}
void sortEdges() {
    for (int i = 0; i < E - 1; i++) {
        for (int j = 0; j < E - i - 1; j++) {
            if (edges[j].weight > edges[j + 1].weight) {
                Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }
}
int find(int[] parent, int i) {
    if (parent[i] == i) return i;
    return find(parent, parent[i]);
}
void union(int[] parent, int[] rank, int x, int y) {
```

```
    int xRoot = find(parent, x);
    int yRoot = find(parent, y);
    if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
    } else if (rank[xRoot] > rank[yRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}
void kruskalMST() {
    Edge[] result = new Edge[V];
    int e = 0;
    int i = 0;
    for (i = 0; i < V; ++i) {
        result[i] = new Edge(0, 0, 0);
    }

    sortEdges();

    int[] parent = new int[V];
    int[] rank = new int[V];
    for (i = 0; i < V; ++i) {
        parent[i] = i;
        rank[i] = 0;
    }
```

```
        i = 0;
        while (e < V - 1) {
            Edge nextEdge = edges[i++];
            int x = find(parent, nextEdge.source);
            int y = find(parent, nextEdge.dest);
            if (x != y) {
                result[e++] = nextEdge;
                union(parent, rank, x, y);
            }
        }
        System.out.println("Minimum Cost Spanning Tree: ");
        int minimumCost = 0;
        for (i = 0; i < e; ++i) {
            System.out.println(result[i].source + " - " + result[i].dest + ": " + result[i].weight);
            minimumCost += result[i].weight;
        }
        System.out.println("Minimum Cost: " + minimumCost);
    }
}
public class prac {
    public static void main(String[] args) {
        int V = 4;
        int E = 5;
        Graph graph = new Graph(V, E);
        graph.edges[0] = new Edge(0, 1, 10);
        graph.edges[1] = new Edge(0, 2, 6);
        graph.edges[2] = new Edge(0, 3, 5);
```

```
        graph.edges[3] = new Edge(1, 3, 15);
        graph.edges[4] = new Edge(2, 3, 4);
        graph.kruskalMST();
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Minimum Cost Spanning Tree:
2 - 3: 4
0 - 3: 5
0 - 1: 10
Minimum Cost: 19
PS D:\Probin's Work\Extra>
```

Conclusion: From this practical I learned how to use greedy approach for getting the best optimal solution.

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 5

Aim: Dynamic Programming Approach

5.1) Let S be a collection of objects with profit-weight values. Implement the 0/1 knapsack problem for S assuming we have a sack that can hold objects with total weight W.

Program Code:

```
import java.util.*;
class ProfitWeight {
   int profit;
   int weight;

   public ProfitWeight(int profit, int weight) {
      this.profit = profit;
      this.weight = weight;
   }
}
public class prac {
   public static int knapsack(ProfitWeight[] objects, int W) {
      int n = objects.length;
      int[][] dp = new int[n + 1][W + 1];
      for (int i = 1; i <= n; i++) {
         for (int j = 0; j <= W; j++) {
            if (objects[i - 1].weight <= j) {
               dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - objects[i - 1].weight] + objects[i -
1].profit);
            } else {
```

```
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][W];
}
public static void main(String[] args) {
    ProfitWeight[] objects = {
        new ProfitWeight(60, 10),
        new ProfitWeight(100, 20),
        new ProfitWeight(120, 30)
    };
    int W = 50;
    System.out.println("Maximum profit: " + knapsack(objects, W));
}
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Maximum profit: 220
PS D:\Probin's Work\Extra>
```

5.2) Implement a program to print the longest common subsequence for the following strings.

Test Case	String1	String2
1	ABCDAB	BDCABA
2	EXPONENTIAL	POLYNOMIAL
3	LOGARITHM	ALGORITHM

Program Code:

```
import java.util.*;
public class prac {
    public static String lcs(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();
        int[][] dp = new int[m + 1][n + 1];

        // Build the dp array
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
```

```
        }

        // Build the LCS string
        StringBuilder lcs = new StringBuilder();
        int i = m, j = n;
        while (i > 0 && j > 0) {
            if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                lcs.insert(0, s1.charAt(i - 1));
                i--;
                j--;
            } else if (dp[i - 1][j] > dp[i][j - 1]) {
                i--;
            } else {
                j--;
            }
        }
        return lcs.toString();
    }

    public static void main(String[] args) {
        String s1 = "ABCDAB";
        String s2 = "BDCABA";
        System.out.println("Longest Common Subsequence: " + lcs(s1, s2));
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Longest Common Subsequence: BDAB
PS D:\Probin's Work\Extra>
```

5.3) Given a chain < A1, A2,...,An> of n matrices, where for i=1,2,...,n matrix Ai with dimensions. Implement the program to fully parenthesize the product A1,A2,...,An in a way that minimizes the number of scalar multiplications. Also calculate the number of scalar multiplications for all possible combinations of matrices.

Test Case	n	Matrices with dimensions
1	3	A1: 3*5, A2: 5*6, A3: 6*4
2	6	A1: 30*35, A2: 35*15, A3: 15*5, A4: 5*10, A5: 10*20, A6: 20*25

Program Code:

```
import java.util.*;
public class prac {

    public static int matrixChainOrder(int[] dimensions) {
        int n = dimensions.length - 1;
```

```
        int[][] dp = new int[n][n];

        for (int i = 0; i < n; i++)
            dp[i][i] = 0;

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i < n - len + 1; i++) {
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    int cost = dp[i][k] + dp[k + 1][j] + dimensions[i] * dimensions[k + 1] *
dimensions[j + 1];
                    if (cost < dp[i][j])
                        dp[i][j] = cost;
                }
            }
        }
        return dp[0][n - 1];
    }

    public static void main(String[] args) {
        int[] dimensions1 = {3, 5, 6, 4};
        System.out.println("Minimum scalar multiplications: " +
matrixChainOrder(dimensions1)+ " for example1");

        int[] dimensions2 = {30, 35, 15, 5, 10, 20, 25};
        System.out.println("Minimum scalar multiplications: " +
matrixChainOrder(dimensions2)+" for example2");
```

```
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
Minimum scalar multiplications: 162 for example1
Minimum scalar multiplications: 15125 for example2
PS D:\Probin's Work\Extra>
```

Conclusion: From this practical I learned about the concept of Dynamic Programming.

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 6

Aim: Backtracking & Branch & Bound

6.1) You are given an integer N. For a given N x N chessboard. Implement a program to find a way to place 'N' queens such that no queen can attack any other queen on the chessboard.

A queen can be attacked when it lies in the same row, column, or the same diagonal as any of the other queens. You have to print one such configuration.

Program Code:

```
import java.util.*;
import java.util.Queue;
import java.time.*;
import java.lang.*;
import java.io.*;
public class prac {
    private static boolean isSafe(int[][] board, int row, int col, int N) {
        int i, j;
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 1)
                return false;
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
```

```
            return false;

        return true;
    }

    private static boolean solveNQueensUtil(int[][] board, int col, int N) {
        if (col >= N)
            return true;
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col, N)) {
                board[i][col] = 1;

                if (solveNQueensUtil(board, col + 1, N))
                    return true;

                board[i][col] = 0;
            }
        }
        return false;
    }

    public static void solveNQueens(int N) {
        int[][] board = new int[N][N];

        if (!solveNQueensUtil(board, 0, N)) {
            System.out.println("Solution does not exist");
            return;
```

```
        }

        printSolution(board, N);
    }

    private static void printSolution(int[][] board, int N) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++)
                System.out.print(board[i][j] + " ");
            System.out.println();
        }
    }
     public static void main(String[] args) {
        int N = 4;
        solveNQueens(N);
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.java
PS D:\Probin's Work\Extra> java prac
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
PS D:\Probin's Work\Extra>
```

6.2) Amar takes 2, 6 and 7 hours of time to perform cooking,

gardening and cleaning respectively. Akbar takes 4, 8 and 3 hours of time to perform cooking, gardening and cleaning respectively. Anthony takes 9, 5 and 1 hours of time to perform cooking, gardening and cleaning respectively. Find out optimal job assignment for Amar, Akbar and Anthony.

Program Code:

```
import java.util.ArrayList;
import java.util.List;

public class prac {
   private static int[][] costs = {
         {2, 6, 7},
         {4, 8, 3},
         {9, 5, 1}
   };
    static int N = 3;
    static boolean[] assigned = new boolean[N];
    static int[] assignment = new int[N];
    static int[] minCostAssignment = new int[N];
    static int minCost = Integer.MAX_VALUE;

   private static void solve(int[] currentAssignment, int currentCost, int currentJob) {
      if (currentCost >= minCost)
         return;

      if (currentJob == N) {
```

```
            minCost = currentCost;
            System.arraycopy(currentAssignment, 0, minCostAssignment, 0, N);
            return;
        }

        for (int i = 0; i < N; i++) {
            if (!assigned[i]) {
                assigned[i] = true;
                currentAssignment[currentJob] = i;
                solve(currentAssignment, currentCost + costs[i][currentJob], currentJob + 1);
                assigned[i] = false;
            }
        }
    }

    public static void main(String[] args) {
        int[] currentAssignment = new int[N];
        solve(currentAssignment, 0, 0);

        System.out.println("Optimal Job Assignment:");
        for (int i = 0; i < N; i++) {
            System.out.println("Amar -> Job " + (minCostAssignment[i] + 1) + " (Cost: " +
costs[minCostAssignment[i]][i] + " hours)");
        }
        System.out.println("Total Cost: " + minCost + " hours");
    }
}
```

Output:

```
PS D:\Probin's Work\Extra> javac prac.jav
PS D:\Probin's Work\Extra> java prac
Optimal Job Assignment:
Amar -> Job 1 (Cost: 2 hours)
Amar -> Job 3 (Cost: 5 hours)
Amar -> Job 2 (Cost: 3 hours)
Total Cost: 10 hours
PS D:\Probin's Work\Extra>
```

Conclusion: From this practical I learned about the branch and bound approach.

Staff Signature:

Grade:

Remarks by the Staff:

Practical – 7

Aim: String Matching

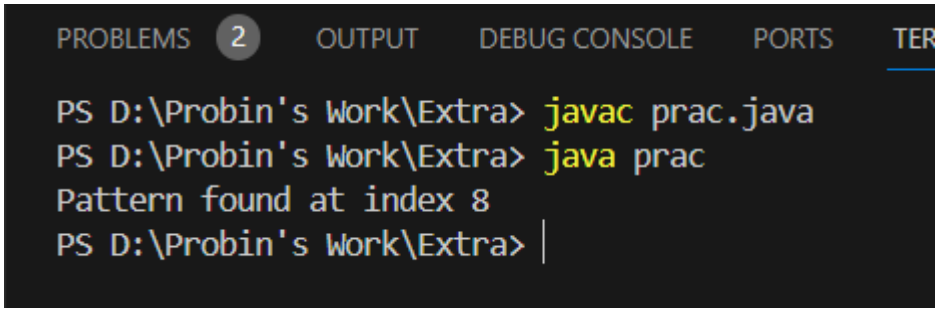
7.1)Two strings, a pattern 'P' and a text 'T' are given. The task is to implement program to determine if the pattern occurs in the text using Rabin Karp algorithm, and if it does, print all of its occurrences; else, print -1.

Program Code:

```
import java.util.ArrayList;
import java.util.List;
public class prac
{
    static int d = 256;
    static int q = 101;
    public static void search(String pattern, String text) {
      int M = pattern.length();
      int N = text.length();
      int i, j;
      int p = 0;
      int t = 0;
      int h = 1;

      for (i = 0; i < M - 1; i++)
         h = (h * d) % q;
      for (i = 0; i < M; i++) {
         p = (d * p + pattern.charAt(i)) % q;
         t = (d * t + text.charAt(i)) % q;
```

```
        }

        for (i = 0; i <= N - M; i++) {
            if (p == t) {
                for (j = 0; j < M; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j))
                        break;
                }
                if (j == M)
                    System.out.println("Pattern found at index " + i);
            }

            if (i < N - M) {
                t = (d * (t - text.charAt(i) * h) + text.charAt(i + M)) % q;
                if (t < 0)
                    t = (t + q);
            }
        }
    }
    public static void main(String[] args) {
        String text = "DFAGBNHDAABAQWER";
        String pattern = "AABA";
        search(pattern, text);
    }
}
```

Output:



Conclusion: From this practical I learned about the string matching using Rabin-Karp Algorithm.

Staff Signature:

Grade:

Remarks by the Staff: